Adoption of academic tools in open source communities: the Debian case study*

Pietro Abate[1], Roberto Di Cosmo[2],

[1] IRILL and INRIA,pietro.abate@inria.fr
[2] INRIA and University Paris Diderot, roberto@dicosmo.org

Abstract. Component repositories play a key role in the open software ecosystem. Managing the evolvution of these repositories is a challenging task, and maintainers are confronted with a number of complex issues that need automatic tools to be adressed properly.
In this paper, we present an overview of 10 years of research in this field and the process leading to the adoption of our tools in a FOSS community. We focus on the Debian distribution and in particular we look at the issues arising during the distribution lifecycle: ensuring buildability of source packages, detecting packages that cannot be installed and bootstrapping the distribution on a new architecture. We present three tools, *distcheck*, *buildcheck* and *botch*, that we believe of general interest for other open source component repositories.
The lesson we have learned during this journey may provide useful guidance for researchers willing to see their tools broadly adopted by the community.

1 Introduction

In the last two decades, component repositories have played an important role in many areas, from software distributions to application development. All major Free and Open Source Software (FOSS) distributions are organized around large repositories of software components. Debian, one of the largest coordinated software collections in history [12], contains in its development branch more than 44'000 binary *packages*[3] generated from over 21'000 source packages; the Central Maven repository has a collection of 100'000 Java *libraries*; the Drupal web framework counts over 16'000 *modules*.

In Debian, components are developed independently by different communities and assembled in the main repository, giving raise to a small world dependency graph [15]. Apart from intrinsic coordination problems associated to this distributed development model, the number of dependencies in Debian distributions poses new challenges for automation and quality assurance. During the last 10 years we have participated in the development and adoption of automatic tools for testing, integration and tracking of all components and aspects of a repository, in particular in the framework of the European project Mancoosi [1].

* Work partially performed at, and supported by IRILL `http://www.irill.org`.
[3] Debian software components are called packages

It is well known that achieving real world adoption of tools developed in academia and proposed by researchers is a painful and difficult process that only rarely succeeds [19]. After years of work, we managed to get almost all of our tools adopted in the Debian project.

We participated in extensive work performed by a team that spent 10 years of research in quality assurance, and package management, an area for which a comprehensive short survey is available elsewhere [8]. During this time, we had different collaborations with many other communities such as the Eclipse [16, 17] and the OCaml [2] with different degrees of success.

In this article, we sum up and share the lessons we have learned in collaboration specifically with the Debian community, because of the direct involvement of a few members of our team, and because of the open and community driven bazaar-style development model. We truly believe that FOSS distribution and software collections alike can benefit from our experience and researcher should invest time and energy to work with developers in a proactive way and foster integration of modern and automatic QA (quality assurance) tools.

The rest of the paper is organised as follows: After a brief introduction, we present *distcheck* and *buildcheck*, the main tools developed by our team. Then we will discuss two examples in which our tools play an important role. The first one related to the distribution life cycle (from development to testing, to the stable release). The second is a tool (*botch*) that is used to bootstrap Debian for new hardware platforms. In the last part of the paper we summarize the lessons we have learned in the last 10 years and provide insights for researches and developer communities interested in embarking into a similar journey.

1.1 Packages in the Debian Distribution

Despite different terminologies, and a wide variety of concrete formats, software repositories use similar metadata to identify the components, their versions and their interdependencies. In general, packages have both *dependencies*, expressing what must be satisfied in order to allow for installation of the package, and *conflicts* that state which other packages must *not* be installed at the same time. As

```
Package: ant
Version: 1.9.7-2~bpo8+1
Installed-Size: 2197
Architecture: all
Depends: default-jre-headless | java5-runtime-headless | java6-runtime-headless
Recommends: ant-optional
Suggests: ant-doc, ant-gcj, default-jdk | java-compiler | java-sdk
Conflicts: libant1.6-java
Breaks: ant-doc (<= 1.6.5-1)
Description: Java based build tool like make
```

Fig. 1. Excerpt of Debian package metadata

shown in Figure 1, while conflicts are simply given by a list of offending packages, dependencies may be expressed using logical conjunction (written '`,`') and disjunctions ('`|`'). Furthermore, packages mentioned in inter-package relations may be qualified by constraints on package versions. Debian packages and come in two flavours: binary packages contain the files to be installed on the end user machine, and source packages that contain all of files needed to build these binary packages. Debian package meta-data describe a broad set of inter-package relationships: virtual-packages, dependencies, multi-architecture annotations, and many more, allow the Debian project to automatize tasks such as binary package recompilations, package life cycle management among different releases, or bootstrapping the distribution on new architectures.

1.2 The installability problem

Finding a way to satisfy all the dependencies of a given package only using the components available in a repository, also known as the *installability problem*, is the key task for all component based repositories: all package managers need to tackle it, be it for Eclipse plugins, Drupal modules, or Debian packages.

And yet, it was not until 2006 that it was shown that this problem is NP-complete for the Debian distribution [7], and later on for a broad range of component repositories [4]. This result came as a kind of a surprise in the different engineering communities, that were using on a daily basis ad-hoc tools which were fast, but under closer scrutiny turned out to be incomplete [7].

Luckily, real world instances proved to be tractable, and it was possible to design and implement dependency solvers based on sound scientific basis, that could significantly outperform all the pre-existing tools: Jerôme Vouillon's early prototypes, `debcheck` and `rpmcheck`, originally developed in 2006, paved the way to modern dependency checking, and are nowadays at the core of the tools we describe in the rest of this paper.

1.3 The Edos and Mancoosi research projects

Edos and Mancoosi [1] are two research projects funded by the European Commission, that run respectively from 2004 to 2007 and from 2008 to 2011. They focused on the new research problems posed by the maintenance of free software distributions, and brought together industries and top research laboratories from over 10 countries. Besides publishing research articles, these projects produced several tools that significantly improved the state of the art, and that are now part of the Dose3 library, which has outlived the research projects and became over time a collection of all the algorithms and tools developed over more than a decade. Unlike what seems to often happen in these research areas [19], most of the tools that were developed have now been adopted, in particular in the Debian distribution, even if with varying degrees of delay and effort.

2 Our Tools

The first two tools produced by this research effort that were adopted in the Debian community are *distcheck* and *buildcheck*, which scan all the packages in a Debian distribution to identify installability issues. Both tools were developed to provide proof of concept prototypes to support our experiments but evolved, with the help of the Debian community, to production ready tools.

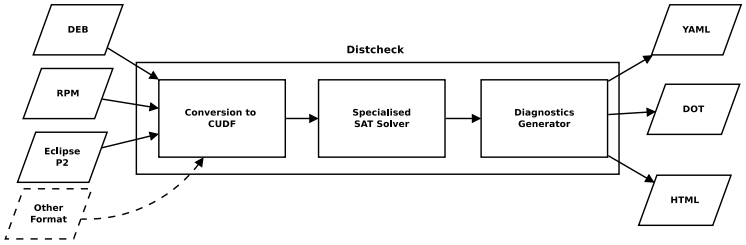

Fig. 2. *distcheck* architecture

2.1 Distcheck and Buildcheck

The *distcheck* tool is a command line tool, capable of verifying the installability of all (or a selection of) components contained in a given repository. Internally, *distcheck* is designed as a pipeline, as shown in Figure 2. The front-end on the left is a multiplexer parser that supports several formats for component metadata (Debian `Packages` files, RPM's synthesis or hdlist files, Eclipse OSGI metadata, etc). After metadata parsing, component inter-relationships are converted in a data representation called CUDF (Common Upgradability Description Format), an extensible format, with rigorous semantics [22], designed to describe installability scenarios coming from diverse environments without making assumptions on specific component models, version schemas, or dependency formalisms. CUDF can be serialized as a compact plain text format, which makes it easy for humans to read component metadata, and which facilitates interoperability with other component managers that are not yet supported by *distcheck*.

The actual installability check work is performed by a specialized solver, that uses the SAT encoding [18] and employs an ad hoc customized Davis-Putnam SAT solver [9] by default instead of the many other standalone solvers now available for dependency solving like [6, 10, 13, 20, 14]. Since all computations are performed in-memory and some of the encoding work is shared between all packages, this solver performs significantly faster than a naive approach that would construct a separate SAT encoding for the installability of each package, and then run an off-the-shelf SAT solver on it. For instance, checking installability of all packages of the Debian main repository of the unstable suite (for 53696 packages[4]) takes just 30 seconds on a commodity 64 bit CPU laptop.

[4] Snapshot of the Debian distribution 27/02/2017.

The final component of the pipeline takes the result from the solver and presents it in a variety of human and machine readable formats to the final user. An important feature of *distcheck* is its ability, in case a package is found not installable, to produce a concise human-readable explanation that points to the reasons of the issue in a machine-readable format.

The *buildcheck* tool follows the same pipeline philosophy of distcheck but it is aimed at source packages. It takes a list of source and binary packages and checks if the build dependencies of each source package can be satisfied with the given binary list. *buildcheck* is based on the same algorithm of *distcheck*, but because of different formats and metadata, packages are mangled behind the scenes in an ad-hoc CUDF that can be feed to the solver. The output, as for *distcheck*, is in YAML format and provides a human-readable explanation of the issue.

Adoption. *distcheck* has been adopted in the Debian project thanks to significant commitment on the side of the researchers. In particular, Ralf Treinen and Fabio Mancinelli, on occasion of the 2006 edition of DebConf (the annual meeting of the Debian project), worked on setting up a dedicated web page for use by the Debian Quality Assurance team. That quality dashboard was originally hosted on http://edos.debian.net and evolved over time, incorporating more tools developed later to detect outdated packages [3], and migrated in 2014 to the official Debian Quality Assurance infrastructure, that is now at qa.debian.org/dose/. Our tools are also part of ther projects like *rebootstrap* and bootstrap.debian.net.

3 Enhancing the Debian distribution build process

The Debian life cycle and evolution process is organised around three repositories: *stable*, which contains the latest official release and does not evolve any more (apart for security and critical updates); *testing*, a constantly evolving repository to which packages are added under stringent qualification conditions, and that will eventually be released as the new stable; and *unstable*, a repository in which additions, removals and modifications are allowed under very liberal conditions. A stringent set of requirements, which are formally defined, must be satisfied by packages coming from unstable to be accepted in testing (also known as package migration), and the repository maintainers have responsibility for enforcing them with the help of ad-hoc tools.

From their first release into the Debian repository, packages evolve over time following a well defined process. When a new version of a *source* package S is available, it is fist introduced in unstable (the only point of entry into the distribution), where the corresponding *binary* packages are gradually built. When a binary package is rebuilt, it replaces the previous version (if any), and when all binary packages associated to S are rebuilt, the old version of S is dropped. Building binary packages can be a long process because of compilation errors and broken dependencies. Moreover, because of the iterative nature of this process,

it is sometimes possible to find in unstable several versions of the same source package, and a mixture of binary packages coming from these different versions of the same source.

In order to allow a smooth monitoring of the build process, to keep track of old and new packages in unstable and to handle the transition of packages from unstable to testing, Debian built a powerful internal infrastructure to automatically build and migrate packages from one repository to another.

3.1 Buildd, sbuildd, dose-tools

The Debian autobuilder network is a management system that allows Debian developers to easily add new source packages to the repository and compile all associated binary package for all architectures currently supported by Debian. This network is made up of several machines and uses a specific software package called *buildd* whose main function is to automatically build all binary packages, according to its metadata and multi-architecture annotations.

The build daemon, consists of a set of scripts (written in Perl and Python) that have evolved to help porters with various tasks. Over time, these scripts have become an integrated system that is able to keep Debian distributions up-to-date nearly automatically. The build infrastructure is composed of three main components, first *wanna-build* is a tool to collect and keep track of all package metadata. The *buildd* is the multiplexer that selects which builder for each architecture must be invoked for each package, and finally *sbuild* is the actual builder to automatically recompile the package. *buildcheck* and *distcheck* are integrated in different components of the Debian build daemon.

buildcheck is used in the *wanna-build* daemon to check if a all the build dependencies of a given source package are available. This step allows one to catch dependency problems before even allowing the package to enter the build queue, hence saving considerable resources and space. *buildcheck* is fed with the metadata of the current source package, and the metadata of all available packages in the archive at one moment in time. By using different options, *buildcheck* is able to check, for each architecture, if the all dependencies are available and if this is not the case, to provide a human readable explanation for the package maintainer. This tool is also available to the package maintainer and it can be run independently on a personal machine.

distcheck is used in *sbuildd* to provide better explanations to the package maintainer in case of failure. Depending on the solver (by default aspcud [11]), the dose3 explainer might report a dependency situation as satisfiable even if the apt-get found it to be unsatisfiable. This is a consequence of the fact that the default Debian resolver (apt-get) employs an algorithm that is, albeit very fast, incomplete. Having a sound and complete dependency solver for Debian helped developers in many occasions. The same solver is also available to the final user via apt-get, where the user can choose to select an external solver while installing a binary package on their machine. Before the introduction of *distcheck* packages where tested for broken dependencies using apt, that because of its nature, it was less adapted to this task.

Adoption. *buildcheck* and *distcheck* have also been adopted in the Debian project thanks to significant commitment on the side of the researchers: in particular, after a common presentation with Ralf Treinen in DebConf 2008 [21], Stefano Zacchiroli gave another presentation in DebConf 2009 that motivated the swift integration of the Dose tools in `wanna-build`. This was highly facilitated by the fact that the Dose tools were already properly packaged for Debian, after the work done by Ralf Treinen, and that they had started to be known in the Debian community thanks to the regular participation of the researchers to these events.

4 Bootstrapping Debian on a New Architecture

With new hardware architectures and custom co-processor extensions being introduced to the market on a regular basis, porting Debian to a new architecture not only involves adapting the low-level software layer for a different hardware, but also considering the inter-dependencies among different components and how these can affect the compilation and packaging process. Binary packages and source packages use meta-data to describe their relationships to other components. Bootstrapping a distribution to a new architecture deals with the problem of customizing the software (source packages) for a specific architecture and to instantiate a new set of binary packages that is consistent with the constraints imposed by the new hardware.

Bootstrapping a distribution is the process by which software is compiled, assembled into packages and installed on a new device/architecture without the aid of any other pre-installed software.

The method routinely used in Debian consists in first, the creation (by cross compilation) of a minimal build system, and later the creation of the final set of binary packages on the new device (by native compilation). Cross compiling a small subset of source packages is necessary because an initial minimal set of binary packages must exist in order to compile a larger set of source packages natively. Once a sufficient number of source packages is cross compiled (we call the set of binary packages produced by them a minimal system) new source packages can be compiled natively on the target system. The minimal system is composed of a coherent set of binary packages that is able to boot on the new hardware and to provide a minimal working OS. This minimal set of binary packages contains at the very least an operating system, a user shell and a compiler. This initial selection is generally provided by distribution architects.

4.1 Botch

Botch is a set of tools designed to help porters to refine and complete this selection in a semi-automatic way and to build the rest of the distribution on top of it [5]. Botch is based on the Dose3 library and re-uses many of its components. The main contribution of botch is the ability of providing a compilation order of source packages to gradually rebuild the entire archive. The goal is to break

compilation loops by pruning build dependencies according to special metadata describing compilation *stages*. At each iteration/stage, new binary packages are added to the repository that in turn will allow new source packages to be build.

The development of botch started with an academic collaboration with Johannes Schauer, a student that participated in a Google Summer of Code co-organised with Debian in 2012 [23]. Slowly, from prototype and thanks for the personal investment of the main developer of botch, it evolved from an academic project into an industry-strength tool.

Adoption. Before botch, porting Debian to a new architecture was a long manual process based on the intuition and personal experience. Because of the complex dependency network and inherent recursive nature of the problem (in order to compile a package we need to compile first all the source packages that will generate its build dependencies), it was also particularly error prone.

Hence it came as no surprise to see that it was adopted pretty swiftly: it was not just a matter of improving quality of a distribution, but of saving weeks of hard work. Botch is now referenced in the Debian official page on bootstrapping https://wiki.debian.org/DebianBootstrap and is used regularly.

5 The Technology Transfer Problem

The adoption path of the tools we have surveyed required significant effort and lasted several years. To understand why this was the case, it is important to take a step back and look at the basic principles governing both the FOSS community and the research community.

5.1 Community vs. Academia

The evolution of Debian has imposed the adoption of many different automated tools to handle the continually growing number of packages, developers and users. Historically, all tools belonging today to the Debian infrastructure have evolved independently, often in competition to each other. Because of the Debian governance model, where no central authority takes technical decisions, the adoption of a specific tool has always been left to the so called *do-ocracy*: if you want a particular tool to be used, you need to show its usefulness, integrate it in the infrastructure yourself, and convince others to use it.

As a consequence, the development and acceptance of these tools has always been quite slow because of the *human factor* and often not because of technical objections: once a developer has spent significant time and energy getting his own tool adopted, it is quite natural that they expect high returns in term of their own image in the project. Hence he will not be particularly open to admit that there is an interest in adopting new, more advanced technologies, and one can observe in the debate surprising arguments, like "that's surely a great tool, but you need to rewrite it using programming language A for it to be accepted", where A is a programming language different from the one used in the new tool.

This attitude has often been one of the first reaction we encountered and often the most difficult to overcome.

On the academic side, researchers face a publication pressure that seldom allows them to invest the time required to gain enough traction within this kind of communities. With these constraints, researchers often focus on one community while simply do not have the time to engage others. On top of that, to convince the infrastructure developers to see the "greater good" associated to adapt and use proved and stable solutions spin-off from research projects, one needs to actually produce a tool that is going to work in real-world situations, and not just in the small test cases often used as validation test-beds for academic publication.

Our approach over the year has been to adapt our way of doing research to match the real-world, following "ante litteram" the path highlighted in [19]. Therefore we invested a considerable amount of time to create tools that were able to work with real data, and at the same time use these data as empirical support in our publications. This approach kept us motivated and at the same time proved to be a good return of investment in the long run.

5.2 The Communication Gap

While approaching the Debian community, we faced issues that were sometimes technical in nature, sometimes political, and sometimes even personal. Moreover, after realizing the communication gap between our academic approach and the FOSS communities, we had to learn to speak a new language, and engage the community on their own ground. Researchers often focus exclusively on the effectiveness and correctness on their approach, while forgetting the cost in terms of integration time and learning curve.

The FOSS community is large and diverse. And while everybody has some technical knowledge, adopting a lingo that is too complex to understand can be counter productive. Hackers are more concerned about the results than the mathematics behind a tool, they are concerned about the ease of use, more than the expressive power of a new language. Providing something the community can readily understand, use and modify, in terms of programming language used, development tools, following de-facto standards, can greatly speed up the time of adoption of a new solution.

In our experience, bridging the academia-community gap has been possible only by actively engaging with the community. This involved, on one side, a significant effort to participate in online forums and live conferences: during the years covered in this article, we presented our work in a major European Developer conference (FOSDEM), and invited lead developers to work with us. We greatly benefit from having few members of our research team personally involved in the Community. While this is not always possible and largely dependent on the personal motivation of each team member, having deep ties within Debian helped us greatly to gain trust and respect. We also hosted hacking sprints and provide support for several events. By meeting the community, we tried to reduce this gap and to engage a fruitful and long-standing collaboration.

5.3 Community Driven Open Development

Our next step was to fully open up our development process and welcome different developers from different communities to contribute to our tools. We started this by funding students interested in FOSS and interacting with other researchers that are already active members of the community. The Dose3 library, which has consolidate most of our research work outcomes, has now an active community of developers, it is packaged for all major FOSS distributions and is currently maintained by the first author. To gain acceptance with the community we followed the unix philosophy, providing a lean and powerful command line tool, and an easily parse-able output. We also provided documentation and examples for other languages such as python or perl to foster interoperability and simplify the integration into existing frameworks. During the years we attracted several students interested to work on the project. Two of them developed important components of the library and one of the has now become one of the main contributors.

Finally, we consider that our commitment to handle real-world case studies, with direct applicability in the field, instead of the usual toy examples, proved to be a real important element of success.

5.4 Lesson Learned

From our experience, we can draw the following recommendations for colleges from academia that want to see their tools adopted.

Be proactive Do not wait for the community to reach out to you for help. It is your task to engage developers and publicize your work.

Communication Attending conferences and learning how to frame our work for a specific community is essential.

Engagement Seeking collaboration, hosting events and participating to the development process of a distribution is essential to build trust and ease acceptance.

The extra mile Provide tools and documentation accessible to a wide audience. Make it easy for your tools to be integrated in the existing framework, do not expect others to do it in your place.

Hiring interns, PhD students or post-docs that are interested in free software is a great way of creating connections between the two worlds, and establishing trust.

6 Conclusion

The take-away from this paper is that developing amazing and efficient tools behind the high walls of academia is only the starting point, and much more is needed to achieve impact in the real world.

Acknowledgements. The work described in this article has been performed over a very long span of time, in collaboration with many researchers, that contributed in different periods, and to varying degrees, to some or all of the tools that we mention. Roberto Di Cosmo, Fabio Mancinelli, Ralf Treinen and Jerôme Vouillon were actively involved in the EDOS project, with Fabio Mancinelli leaving after that period. In the MANCOOSI project, that was set up and coordinated by Roberto Di Cosmo, Ralf Treinen and Jerôme Vouillon were joined by Pietro Abate, Jaap Boender and Stefano Zacchiroli.

References

1. The mancoosi project. http://www.mancoosi.org, 2011.
2. P. Abate, R. D. Cosmo, L. Gesbert, F. L. Fessant, R. Treinen, and S. Zacchiroli. Mining component repositories for installability issues. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, pages 24–33, 2015.
3. P. Abate, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Learning from the future of component repositories. *Sci. Comput. Program.*, 90, 2014.
4. P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10), October 2012.
5. P. Abate and J. Schauer. Bootstrapping software distributions. In *Proceedings of Itnl Symposium of Component Based Software Engineering CBSE*, 2013.
6. J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault. Solving Linux upgradeability problems using boolean optimization. In *LoCoCo: Logics for Component Configuration*, volume 29 of *EPTCS*, 2010.
7. R. Di Cosmo, F. Mancinelli, J. Boender, J. Vouillon, B. Durak, X. Leroy, D. Pinheiro, P. Trezentos, M. Morgado, T. Milo, T. Zur, R. Suarez, M. Lijour, and R. Treinen. Report on formal mangement of software dependencies. Technical report, EDOS, Apr. 2006.
8. R. Di Cosmo, R. Treinen, and S. Zacchiroli. Formal aspects of free and open source software components - a short survey. In *FMCO*, pages 216–239, 2012.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, 2004.
10. J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. Towards automated analysis. In *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications*. CEUR-WS.org, 2010.
11. M. Gebser, R. Kaminski, and T. Schaub. aspcud: A Linux package configuration tool based on answer set programming. In C. Drescher, I. Lynce, and R. Treinen, editors, *Proceedings Logics for Component Configuration, LoCoCo*, 2011.
12. J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3), 2009.
13. M. Janota. Do sat solvers make good configurators? In *SPLC: Software Product Lines Conference, 2nd Volume*, 2008.

14. G. Jenson, J. Dietrich, and H. Guesgen. An empirical study of the component dependency resolution search space. In *CBSE 2011: International ACM Sigsoft Symposium on Component Based Software Engineering*, volume 6092 of *LNCS*. Springer, 2010.
15. N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.
16. D. Le Berre and A. Parrain. On SAT technologies for dependency management and beyond. In *SPLC 2008: Software Product Lines Conference, 2nd Volume*, 2008.
17. D. Le Berre and P. Rapicault. Dependency management for the Eclipse ecosystem. In *IWOCE 2009: International Workshop on Open Component Ecosystems*. ACM, 2009.
18. F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006: Automated Software Engineering*. IEEE, 2006.
19. R. Marinescu. Confessions of a worldly software miner. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015.
20. C. Michel and M. Rueher. Handling software upgradeability problems with MILP solvers. In *LoCoCo 2010: Logics for Component Configuration*, volume 29 of *EPTCS*, 2010.
21. R. Treinen and S. Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *DebConf 8: proceedings of the 9th conference of the Debian project*, 2008.
22. R. Treinen and S. Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, Nov. 2009.
23. Wookey and P. Abate. Google summer of code on debian bootstrap. 2012.